Intro to R – Part II

R for Stata Users

Luiza Andrade, Leonardo Viotti & Rob Marty

April, 2019

Outline

Why are we here today?

- In the last session, you learned the basic concepts to work in R
- You are probably eager to get your hands into some data using R by now, and you would figure out *what* should be in your code for it to work
- But you would probably not know right away *how* to write that, so that in the end you might have code that is only intelligible for yourself – and not for a very long time

Why are we here today?

- In this session, we will cover common coding practices in R so that you can make the most efficient use for it
- We will also discuss some styling conventions to make your code readable and reproducible
- This will give you a solid foundation to code in R, and hopefully you'll be able to skip some painful steps of the "getting-your-hands-dirty" learning approach

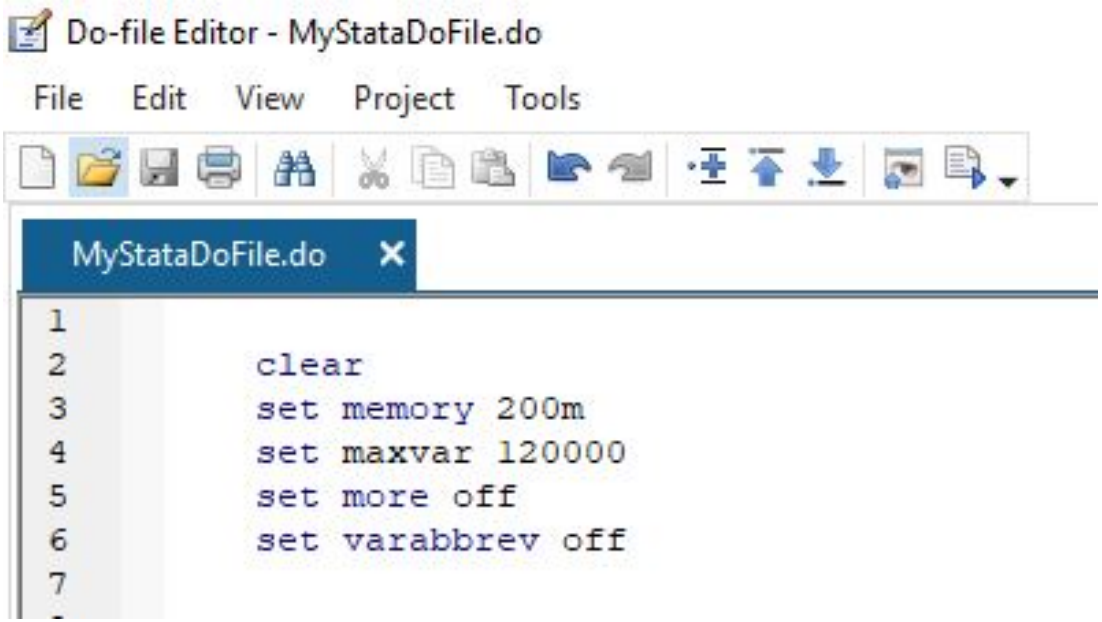
Outline

Initial Settings

- Ok, let's start by opening RStudio
- What do you see in your environment?
- If you saved the last RStudio session in `.Rhistory` (and that is the default), the objects that were in RStudio's memory last time you closed it will still be there whenever you open it again

Initial Settings

Raise your hand if you have ever seen these lines of code before:



Initial Settings

- We don't need to set the memory or the maximum number of variables in R, and the equivalent of `more` is the default in R
- You can see all the objects currently in you memory in the *Environment* pane

Initial Settings

Exercise 1: Clear workspace

- Make sure the `Environment` window is open. Is anything there?
- Create an object called `foo` with any content you pick
- Type `rm(foo)` to remove the `foo` object from you memory
- Type `ls()` to print the names of the object in memory
- To remove all objects, use `rm(list=ls())`

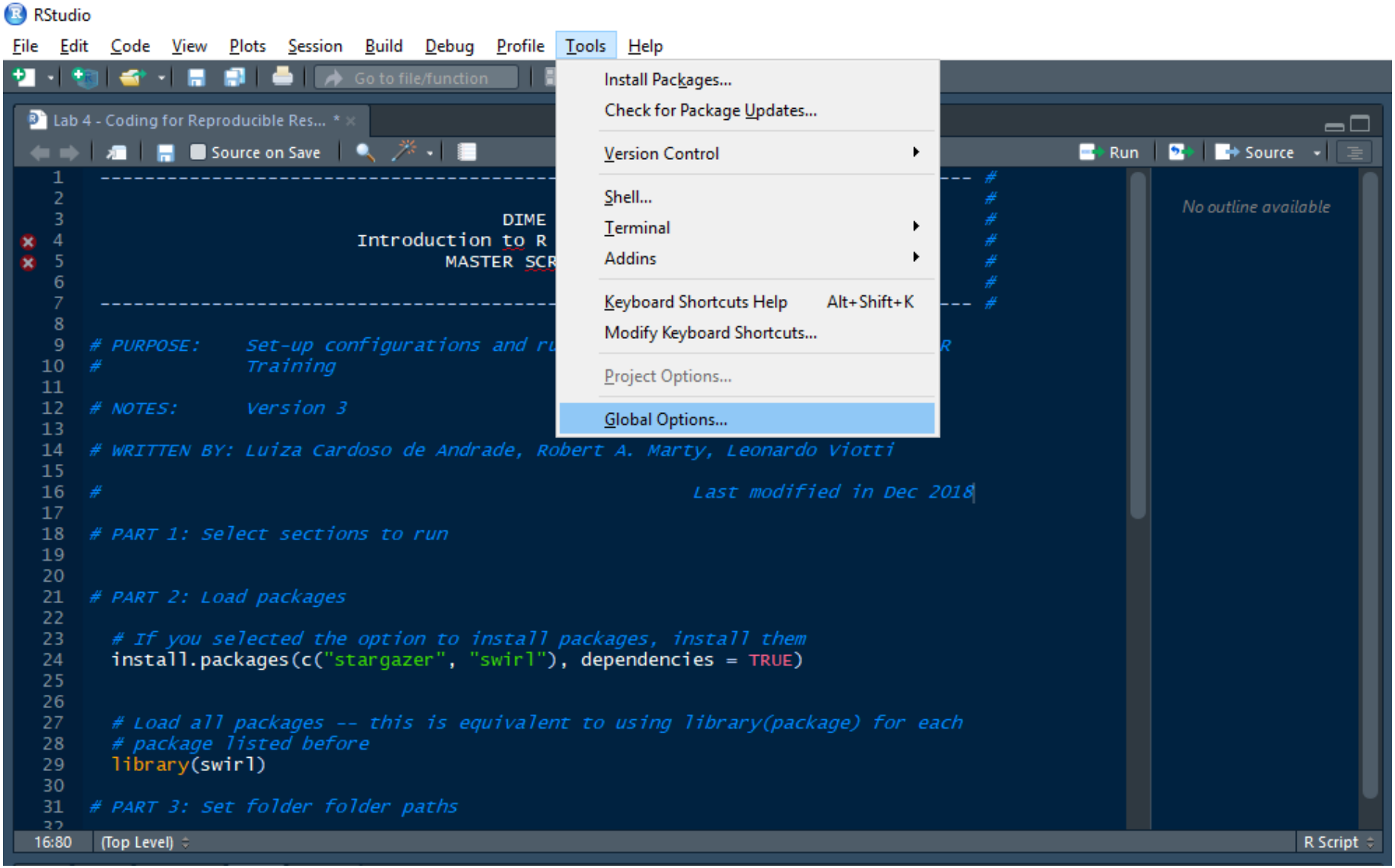
Initial Settings

Initial Settings

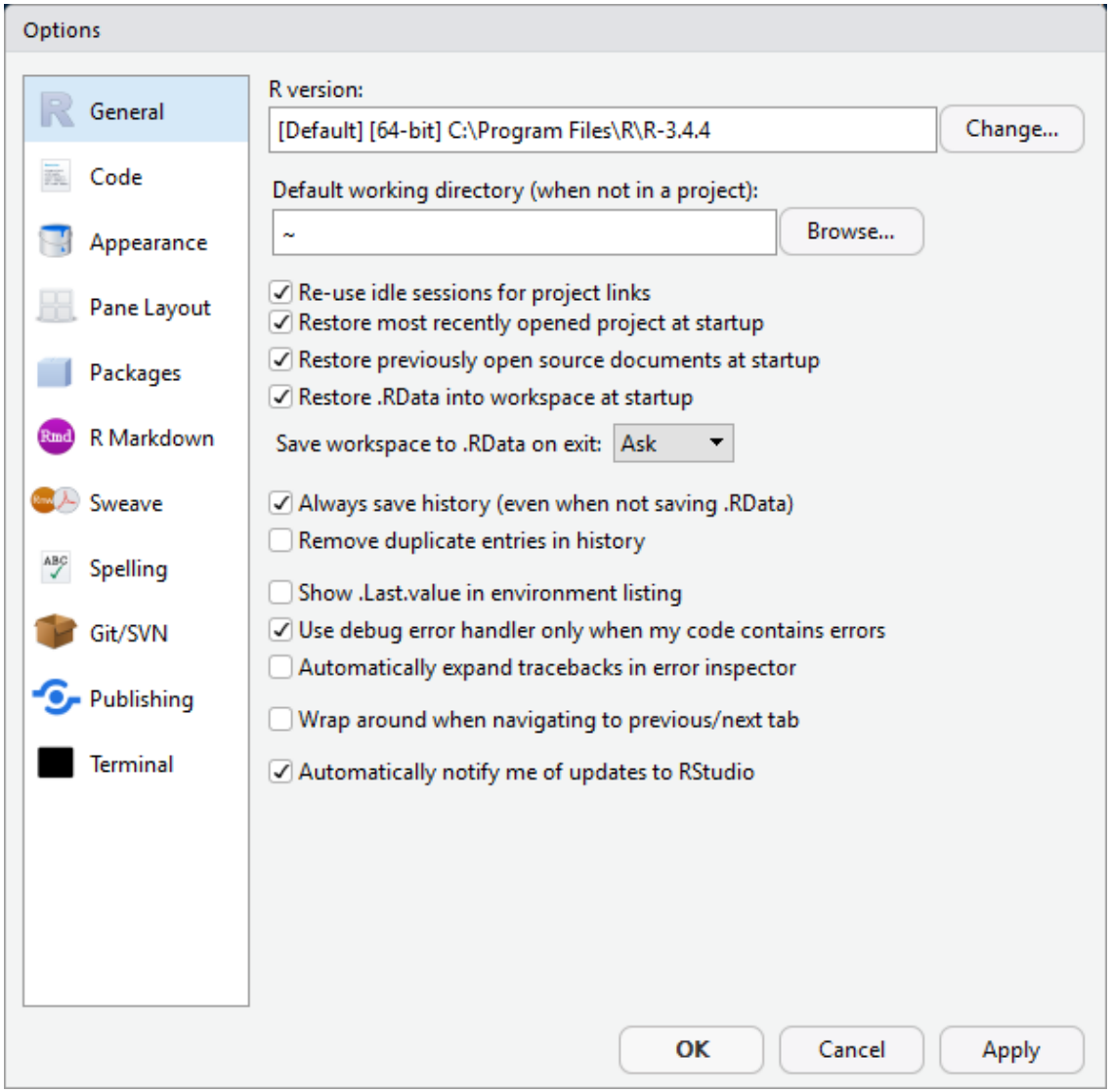
Exercise 2: No one will burn your computer

Here's how you change these settings

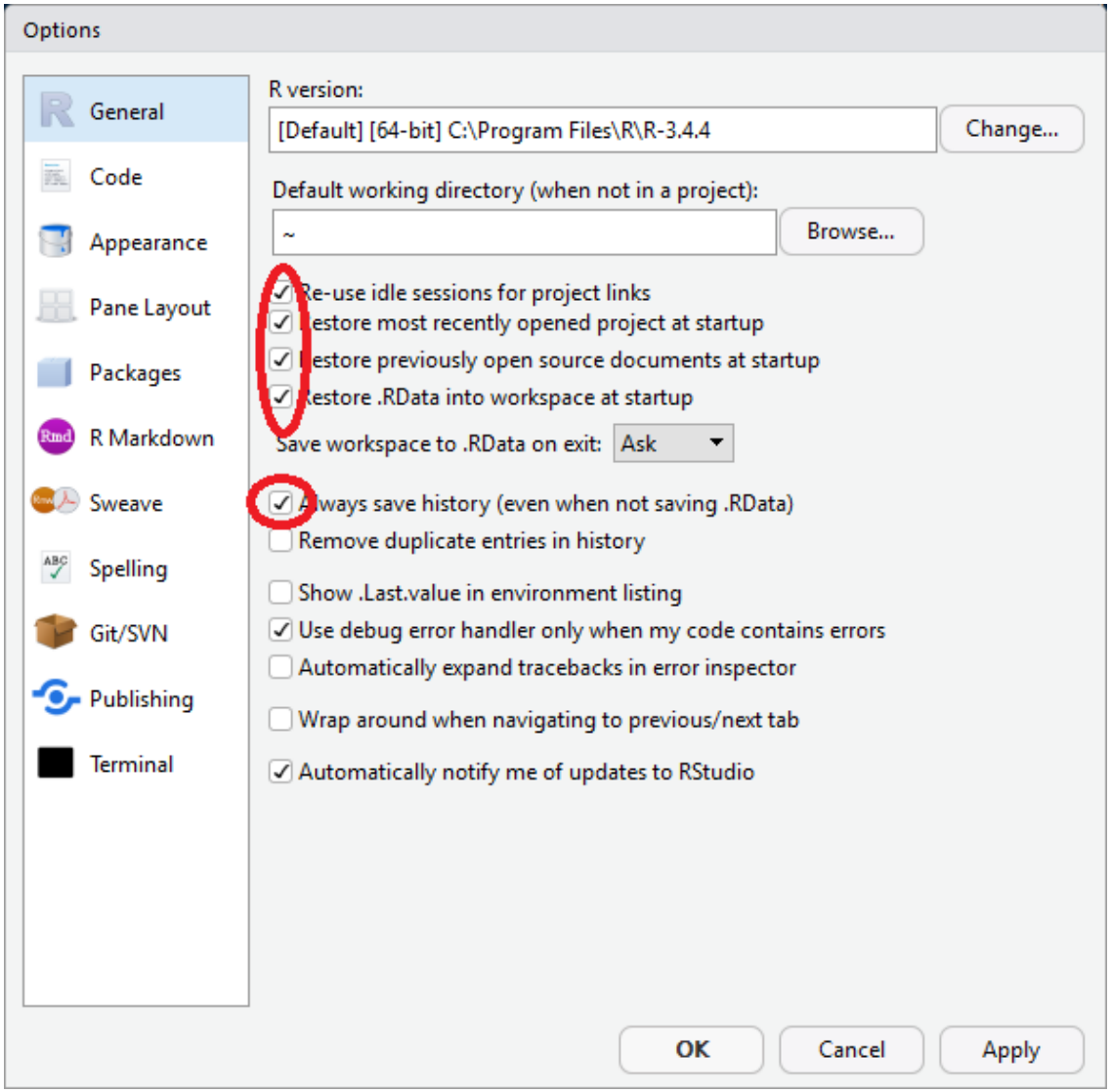
Initial Settings



Initial Settings



Initial Settings



File paths

- For the purpose of this training, we will assume that you are dealing with a specific folder structure
- Folder organization, though an important part of data work, is outside the scope of this course
- You can find resources about it in the appendix, and we have shared with you a folder that is organized as we want it to be
- To follow today's session, go to the `DataWork/Code` folder and open the file called `Lab 2 - Intro II.R`
- We will use this script as a basis for the exercises, and you should modify it during this session to complete them

Outline

File paths

- In the last session, we used the menu bar to load a data set into R
- Today, we will do that using code and referring a file's path
- File paths in R, as in Stata, are basically just strings
- Note, however, that in R we can only use forward slashes (/) to separate folder names
- We recommend always using **explicit** and **dynamic** file paths

File paths

- Explicit and dynamic file path:

```
# Define dynamic file path
finalData <- "C:/Users/luiza/Documents/GitHub/
              dime-r-training/
              DataWork/DataSets/Final"

# Load data set
whr <- read.csv(file.path(finalData,"whr_panel.csv"),
                header = T)
```

Exercise 3: File path to your folder

Let's start by adding the folder path to the training's folder in your computer to the beginning of `PART 2`

File paths

- You can set file paths in your script using the `file.path()` function
- This function concatenates strings using `/` as a separator to create file paths

File paths

Let's test if that worked:

```
  # Project folder
  projectFolder  <-
    "C:/Users/luiza/Documents/GitHub/dime-r-training"

  # Data work folder
  dataWorkFolder    <- file.path(projectFolder,"DataWork")

  # Print data work folder
  dataWorkFolder
```

```
## [1] "C:/Users/luiza/Documents/GitHub/dime-r-training/DataWork"
```

Loading a data set from CSV

`read.csv(file, header = FALSE)`

- **file**: is the path to the file you want to open, including it's name and format (`.csv`)
- **header**: if `TRUE`, will read the first row as variable names
- **stringsAsFactors:** logical. See next slide for more.

Loading a data set from CSV

- R reads string variables as factors as default
- This format saves memory, but can be tricky if you actually want to use the variables as strings
- You can specify the option `stringsAsFactors = FALSE` to prevent R from turning strings into factors

Loading a data set from CSV

Exercise 4: Test file paths

1. Save your code.
2. Start a new R session: go to `Session > New Session`. This session should be completely blank.
3. Open the code you just saved.
4. Add a line opening the data set in `PART 5` of your Master script

```
# Load data set
whr <- read.csv(file.path(finalData,"whr_panel.csv"),
                header = T)
```

5. Run the whole script. If it worked, your environment should include only the `whr` data set.

Outline

Exploring a data set

Some useful functions:

- `View()`: open the data set
- `class()`: reports object type or type of data stored
- `dim()`: reports the size of each one of an object's dimension
- `names()`: returns the variable names of a data set
- `str()`: general information on an R object
- `summary()`: summary information about the variables in a data frame
- `head()`: shows the first few observations in the dataset
- `tail()`: shows the last few observations in the dataset

Exploring a data set

Exercise 5: Explore a data set

Use some of the functions listed above to explore the `whr` data set.

Exploring a data set

```
# View the data set (same as clickin on it in the Environment pane)
View(whr)
```

Exploring a data set

```
class(whr)
```

```
## [1] "data.frame"
```

```
dim(whr)
```

```
## [1] 469  12
```

Exploring a data set

```
str(whr)
```

```
## 'data.frame':    469 obs. of  12 variables:
##  $ country       : Factor w/ 161 levels "","Afghanistan",..: 108 40 60 1
##  $ region        : Factor w/ 11 levels "","Australia and New Zealand",..
##  $ year          : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 201
##  $ happy_rank    : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ happy_score   : num  7.54 7.52 7.5 7.49 7.47 ...
##  $ gdp_pc        : num  1.62 1.48 1.48 1.56 1.44 ...
##  $ family        : num  1.53 1.55 1.61 1.52 1.54 ...
##  $ health        : num  0.797 0.793 0.834 0.858 0.809 ...
##  $ freedom       : num  0.635 0.626 0.627 0.62 0.618 ...
##  $ trust_gov_corr: num  0.316 0.401 0.154 0.367 0.383 ...
##  $ generosity    : num  0.362 0.355 0.476 0.291 0.245 ...
##  $ dystopia_res  : num  2.28 2.31 2.32 2.28 2.43 ...
```

Exploring a data set

```
summary(whr)
```

```
##        country                                  region         year
##  Afghanistan:  3   Sub-Saharan Africa              :115   Min.   :2015
##  Albania    :  3   Central and Eastern Europe      : 87   1st Qu.:2015
##  Algeria    :  3   Latin America and Caribbean     : 70   Median :2016
##  Angola     :  3   Western Europe                  : 63   Mean   :2016
##  Argentina  :  3   Middle East and Northern Africa: 57   3rd Qu.:2017
##  Armenia    :  3   Southeastern Asia               : 26   Max.   :2017
##  (Other)    :451   (Other)                         : 51
##    happy_rank      happy_score        gdp_pc           family
##  Min.   :  1.00   Min.   :2.693   Min.   :0.0000   Min.   :0.0000
##  1st Qu.: 40.00   1st Qu.:4.440   1st Qu.:0.6672   1st Qu.:0.7219
##  Median : 79.00   Median :5.303   Median :1.0279   Median :0.9440
##  Mean   : 78.66   Mean   :5.373   Mean   :0.9641   Mean   :0.9243
##  3rd Qu.:118.00   3rd Qu.:6.239   3rd Qu.:1.2918   3rd Qu.:1.1346
##  Max.   :157.00   Max.   :7.537   Max.   :1.8708   Max.   :1.6106
##
##      health          freedom       trust_gov_corr      generosity
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.00000   Min.   :0.0000
##  1st Qu.:0.3752   1st Qu.:0.2767   1st Qu.:0.05974   1st Qu.:0.1546
##  Median :0.6046   Median :0.4058   Median :0.09858   Median :0.2257
##  Mean   :0.5555   Mean   :0.3835   Mean   :0.13283   Mean   :0.2440
##  3rd Qu.:0.7299   3rd Qu.:0.4955   3rd Qu.:0.17233   3rd Qu.:0.3147
##  Max.   :0.9528   Max.   :0.6582   Max.   :0.50521   Max.   :0.8381
##
##   dystopia_res
##  Min.   :0.3779
##  1st Qu.:1.8198
##  Median :2.1599
##  Mean   :2.1686
##  3rd Qu.:2.5465
```

Exploring a data set

```
head(whr)
```

```
##       country         region year happy_rank happy_score   gdp_pc   family
## 1      Norway Western Europe 2017          1       7.537 1.616463 1.533524
## 2     Denmark Western Europe 2017          2       7.522 1.482383 1.551122
## 3     Iceland Western Europe 2017          3       7.504 1.480633 1.610574
## 4 Switzerland Western Europe 2017          4       7.494 1.564980 1.516912
## 5     Finland Western Europe 2017          5       7.469 1.443572 1.540247
## 6 Netherlands Western Europe 2017          6       7.377 1.503945 1.428939
##      health   freedom trust_gov_corr generosity dystopia_res
## 1 0.7966665 0.6354226      0.3159638  0.3620122     2.277027
## 2 0.7925655 0.6260067      0.4007701  0.3552805     2.313707
## 3 0.8335521 0.6271626      0.1535266  0.4755402     2.322715
## 4 0.8581313 0.6200706      0.3670073  0.2905493     2.276716
## 5 0.8091577 0.6179509      0.3826115  0.2454828     2.430182
## 6 0.8106961 0.5853845      0.2826618  0.4704898     2.294804
```

Exploring a data set

Didn't get all of those? Don't worry, you'll see them again soon.

Outline

Commenting

- To comment a line, write # as its first character
- You can also add # half way through a line to comment whatever comes after it
- In Stata, you can use /* and */ to comment part of a line's code. That is not possible in R: whatever comes after # will be a comment
- To comment a selection of lines, press `Ctrl + Shift + C`

Commenting

Exercise 6: Commenting

1. Go the `Lab 2 - Intro II.R` script. You can see that the first few lines in the script are the header, but they're not commented out.
2. Use the keyboard shortcut to comment the header of the script.
3. Use the keyboard shortcut to comment the header of the script again. What happened?

Outline

Creating a document outline in RStudio

- RStudio also allows you to create an interactive index for your scripts
- To add a section to your code, create a commented line with the title of your section and add at least 4 trailing dashes, pound signs or equal signs after it

Exercise 7: Headers

1. Open the script index and make `PART 1` a section header. Do the same for parts 2 and 3.
2. Note that once you create a section header, an arrow appears right next to it. Click on the arrows of parts 2 and 3 to see what happens.

Creating a document outline in RStudio

- The outline can be accessed by clicking on the button on the top right corner of the script window. You can use it to jump from one section to another
- You can also use the keyboard shortcuts `Alt + L` (`Cmd + Option + L` on Mac) and `Alt + Shift + L` to collapse and expand sections

Outline

Using packages

- Since there is a lot of people developing for R, it can have many different functionalities.
- To make it simpler, these functionalities are bundled into packages.
- A package is just a unit of shareable code.

Using packages

- It may contain new functions, but also more complex functionalities, such as a Graphic User Interface (GUI) or settings for parallel processing (similar to Stata MP).
- They can be shared through R's official repository - CRAN (13,000+ packages reviewed and tested).
- There are many other online sources such as GitHub, but it's important to be careful, as these probably haven't gone through a review process as rigorous as those in CRAN.

Using packages

- To install and use packages you can either do it with the user interface or by the command prompt.

```
# Installing a package
install.packages("tidyverse",
                 dependencies = T)
# the dependencies argument also installs all other packages
# that it may depend upon to run
```

- You only have to install a package once, but you have to load it every new session.

Using packages

Exercise 8

1. Now load the package we just installed. Use the `library()` function to do it.

Using packages

```
library(tidyverse)
```

Warnings vs Errors

What if this happens?

```
> library(tidyverse)
-- Attaching packages ------------------------------------- tidyverse 1.2.1 --
v tibble  2.1.1       v dplyr   0.8.0.1
v readr   1.3.1       v stringr 1.4.0
v purrr   0.3.2       v forcats 0.3.0
-- Conflicts ---------------------------------------- tidyverse_conflicts() --
x dplyr::arrange()      masks plotly::arrange(), plyr::arrange()
x dplyr::combine()      masks gridExtra::combine()
x purrr::compact()      masks plyr::compact()
x dplyr::count()        masks plyr::count()
x raster::extract()     masks tidyr::extract()
x dplyr::failwith()     masks plyr::failwith()
x dplyr::filter()       masks plotly::filter(), stats::filter()
x cowplot::ggsave()     masks ggplot2::ggsave()
x dplyr::id()           masks plyr::id()
x dplyr::lag()          masks stats::lag()
x latticeExtra::layer() masks ggplot2::layer()
x dplyr::mutate()       masks plotly::mutate(), plyr::mutate()
x dplyr::rename()       masks plotly::rename(), plyr::rename()
x dplyr::select()       masks raster::select(), plotly::select()
x dplyr::summarise()    masks plotly::summarise(), plyr::summarise()
x dplyr::summarize()    masks plyr::summarize()
Warning messages:
1: package 'tidyverse' was built under R version 3.4.4
2: package 'tibble' was built under R version 3.4.4
3: package 'readr' was built under R version 3.4.4
4: package 'purrr' was built under R version 3.4.4
5: package 'dplyr' was built under R version 3.4.4
6: package 'stringr' was built under R version 3.4.4
7: package 'forcats' was built under R version 3.4.4
>
```

Warinings vs Errors

R has two types of error messages, `warnings` and actual `errors`:

- `Errors` - break your code, i.e., prevent it from running.
- `Warnings` - usually mean that nothing went wrong yet, but you should be careful.

RStudio's default is to print warning messages, but not stop the code at the lines where they occur. You can configure R to stop at warnings if you want.

Outline

Functions inception

- In R, you can write one function inside another
- In fact, you have already done this a few times in this course
- Here's an example:

Functions inception

```
# Doing it the long way -----------------------------
# Create a vector with the log of the happiness score
log_score <- log(whr$happy_score)

# Get descriptive statistics for the log vector
summary(log_score)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.9907  1.4907  1.6683  1.6580  1.8308  2.0198
```

```
# Shortcut to get to the same place ----------------
summary(log(whr$happy_score))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.9907  1.4907  1.6683  1.6580  1.8308  2.0198
```

Functions inception

- This is a simple example of metaprogramming (that's the real name of this technique) and may seem trivial, but it's not
- For starters, you can't do it in Stata!

Functions inception

```
  ___  ____  ____  ____  ____ (R)
 /__    /   ____/   /   ____/
___/   /   /___/   /   /___/   15.1   Copyright 1985-2017 StataCorp LLC
  Statistics/Data Analysis             StataCorp
                                       4905 Lakeway Drive
     MP - Parallel Edition             College Station, Texas 77845 USA
                                       800-STATA-PC        http://www.stata.com
                                       979-696-4600        stata@stata.com
                                       979-696-4601 (fax)

681-user 4-core Stata network perpetual license:
       Serial number:  501506002486
         Licensed to:  WBG User
                       World Bank Group

Notes:
      1.  Unicode is supported; see help unicode_advice.
      2.  More than 2 billion observations are allowed; see help obs_advice.
      3.  Maximum number of variables is set to 120000; see help set_maxvar.
      4.  New update available; type -update all-

running C:\Program Files (x86)\Stata15\sysprofile.do ...

. sysuse auto
(1978 Automobile Data)

. summarize log(make)
variable log not found
r(111);

.
```

Functions inception

- This is a very powerful technique, as you will soon see
- It's also a common source of error, as you can only use one function inside the other if the output of the inner function is the same as the input of the outer function
- It can also get quite tricky
- Which is why we sometimes use pipes

Functions inception

```r
# Doing it the long way ----------------------------
# Create a vector with the log of the happiness score
log_score <- log(whr$happy_score)

# Get descriptive statistics for the log vector
mean(log_score)

# Shortcut to get to the same place ----------------
mean(log(whr$happy_score))

# Now with pipes -----------------------------------
whr$happy_score %>%
  log() %>%
  mean()
```

Functions inception

You don't need to worry about piping for now, just know that it exists, and laugh if you see this sticker in some tidyverse nerd computer

Outline

Looping

- One thing that usually gives people away as Stata users writing R code are loops
- In Stata, we use for loops quite a lot
- The equivalent to that in R would be to write a for loop like this

```
# A for loop in R
for (number in 1:5) {
    print(number)
}
```

Looping

```
# A for loop in R
for (number in 1:5) {
    print(number)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Looping

- R, however, has a whole function family that allows users to loop through an object in a more efficient way
- They're called `apply` and there are many of them, with different use cases
- If you look for the `apply` help file, you can see all of them
- For the purpose of this training, we will only use two of them, `sapply` and `apply`

Looping

- `sapply(X, FUN, ...)`: applies a function to all elements of a vector or list and returns the result in a vector. Its arguments are
  - **X:** a matrix (or data frame) the function will be applied to
  - **FUN:** the function you want to apply
  - `...`: possible function options

Looping

```
# A for loop in R
for (number in c(1.2,2.5)) {
  print(round(number))
}
```

```
## [1] 1
## [1] 2
```

```
# A much more elegant loop in R
sapply(c(1.2,2.5), round)
```

```
## [1] 1 2
```

Looping

A more general version is the `apply` function.

- `apply(X, MARGIN, FUN, ...)`: applies a function to all columns or rows of matrix. Its arguments are
  - **X:** a matrix (or data frame) the function will be applied to
  - **MARGIN:** 1 to apply the function to all rows or 2 to apply the function to all columns
  - **FUN:** the function you want to apply
  - **`...`:** possible function options

Looping

```
# Create a matrix
matrix <- matrix(c(1, 24, 9, 6, 9, 4, 2, 74, 2),
                 nrow = 3)

# Look at the matrix
matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    6    2
## [2,]   24    9   74
## [3,]    9    4    2
```

Looping

```
    # Row means
    apply(matrix, 1, mean)
```

```
## [1]  3.00000 35.66667  5.00000
```

```
    # Column means
    apply(matrix, 2, mean)
```

```
## [1] 11.333333  6.333333 26.000000
```

Outline

Custom functions

- As you have said several times, R is super flexible
- One example of that is that it's super easy and quick to create custom functions
- Here's how:

Custom functions

```
square <- function(x) {

  y <- x^2

  return(y)

}

square(2)
```

```
## [1] 4
```

Custom functions

Exercise 9: Create a function

Create a function that calculates the z-score of a variable.

Custom functions

```
zscore <- function(x) {

  mean <- mean(x, na.rm = T)
  sd   <- sd(x, na.rm = T)
  z    <- (x - mean)/sd

  return(z)

}

summary(zscore(whr$happy_score))
```

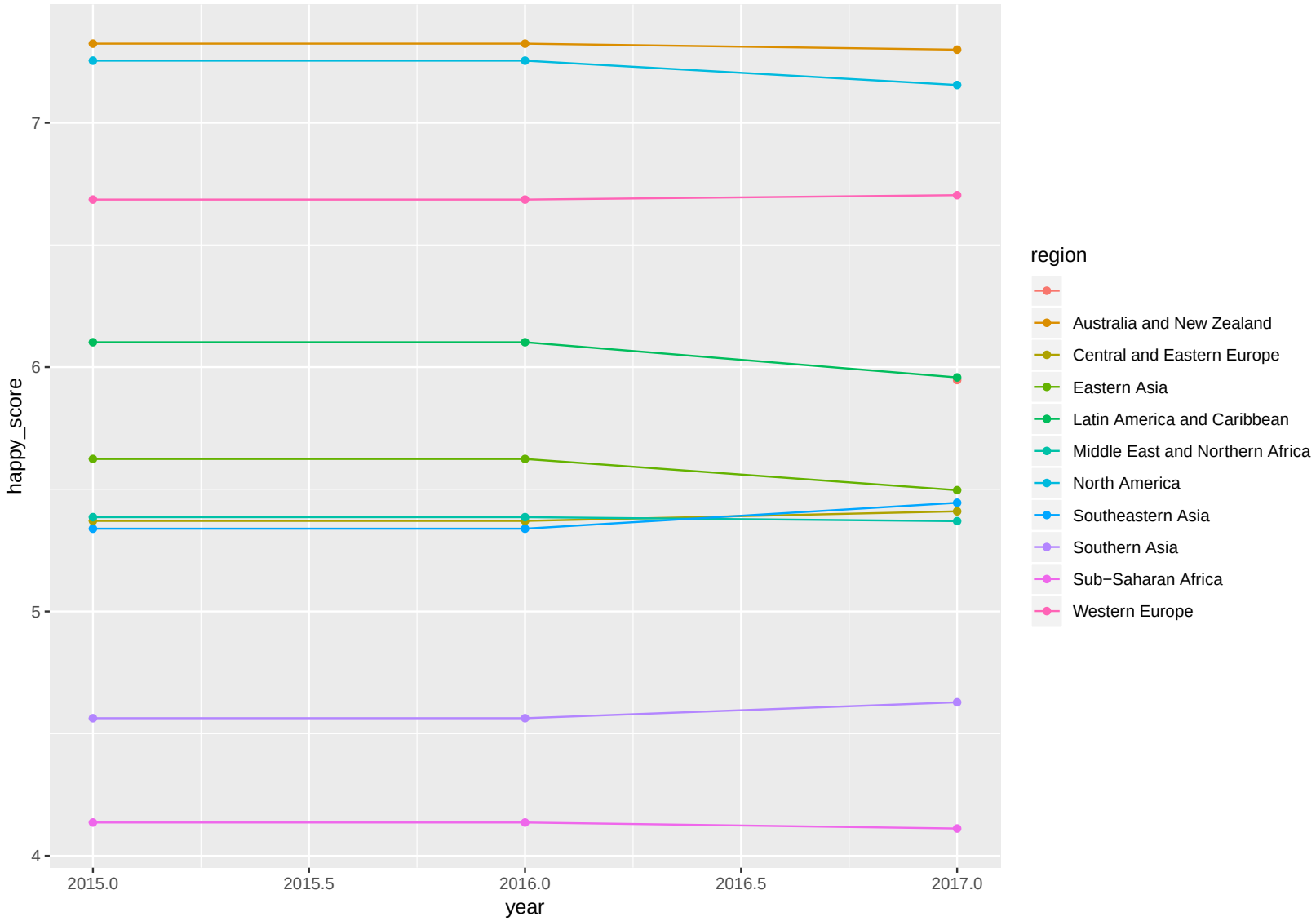
```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -2.35930 -0.82128 -0.06152  0.00000  0.76252  1.90525
```

Outline

Why indent?

```
# Here's some code
annualHappy_reg <- aggregate(happy_score ~ year + region, data = whr, FUN = mean)
plot <- ggplot(annualHappy_reg,aes(y = happy_score,x = year, color = region,
group = region)) + geom_line() + geom_point()
print(plot)
```

Why indent?


happy_score
7
6
5
4
2015.0
2015.5
2016.0
2016.5
2017.0
year
region
Australia and New Zealand
Central and Eastern Europe
Eastern Asia
Latin America and Caribbean
Middle East and Northern Africa
North America
Southeastern Asia
Southern Asia
Sub−Saharan Africa
Western Europe

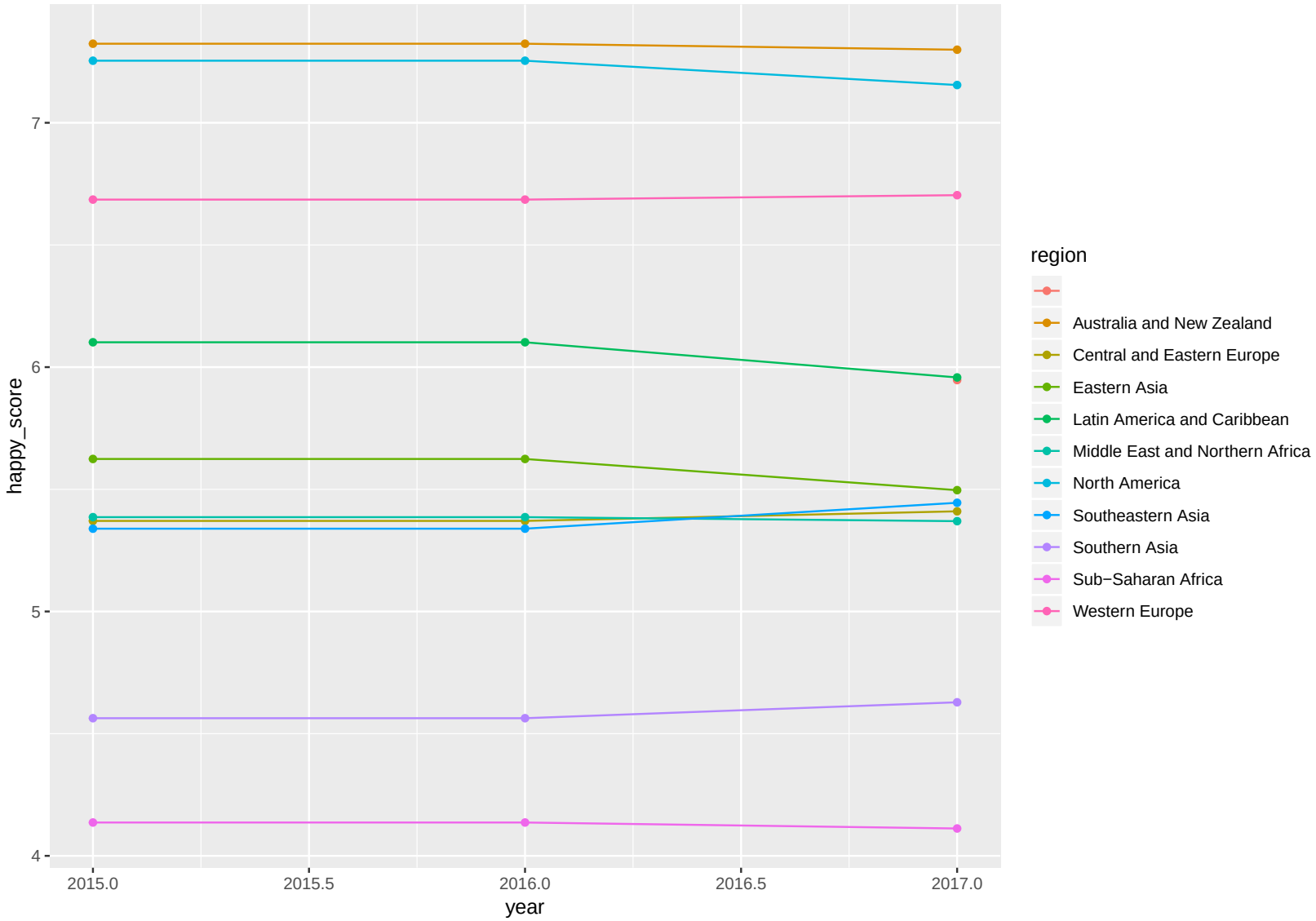
Why indent?

```
# Here's the same code
annualHappy_reg <-
  aggregate(happy_score ~ year + region,
            data = whr,
            FUN = mean)

plot <-
  ggplot(annualHappy_reg,
         aes(y = happy_score,
             x = year,
             color = region,
             group = region)) +
  geom_line() +
  geom_point()

print(plot)
```

Why indent?


happy_score
7
6
5
4
2015.0
2015.5
2016.0
2016.5
2017.0
year
region
Australia and New Zealand
Central and Eastern Europe
Eastern Asia
Latin America and Caribbean
Middle East and Northern Africa
North America
Southeastern Asia
Southern Asia
Sub−Saharan Africa
Western Europe

Why indent?

- Even though R understands what unindented code says, it can be quite difficult for a human being to read it
- On the other hand, white space does not have a special meaning for R, so it will understand code that is more readable for a human being

Indentation

- Indentation in R looks different than in Stata:
  - To indent a whole line, you can select that line and press `Tab`
  - To unindent a whole line, you can select that line and press `Shift + Tab`
  - However, this will not always work for different parts of a code in the same line
- In R, we typically don't introduce white space manually
- It's rather introduced by RStudio for us

Indentation

Exercise 8: Indentation in R

To see an example of how indenting works in RStudio, go back to our first example with `sapply`:

```
# A much more elegant loop in R
sapply(c(1.2,2.5), round)
```

1. Add a line between the two arguments of the function (the vector of numbers and the round function)
2. Now add a line between the numbers in the vector.

Indentation

Note that RStudio formats the different arguments of the function differently:

```
# A much more elegant loop in R
sapply(c(1.2,
         2.5),
       round)
```

The end

Thank you!

Assignment

Create a function that

1. Takes as argument a vector of packages names
2. Loops through the packages listed in the input vector
3. Install the packages
4. Loads the packages

Outline

If statements

- Installing packages can be time-consuming, especially as the number of packages you're using grows, and each package only needs to be installed once
- in the same way we use locals in Stata to create In Stata, section switches would be saved as locals
- In R, the equivalent to that would be to create a new object

Exercise 9: Creating an if statement

Create a dummy scalar object called PACKAGES.

- TIP: Section switches can also be Boolean objects.

If statements

- Now we need to create an if statement using this switch
- If statements in R look like this:

```
# Turn switch on
PACKAGES <- 1

# Install packages
if (PACKAGES == 1) {
  install.packages(packages,
                   dependencies = TRUE)
}
```

If statements

- Possible variations would include

```
# Turn switch on
PACKAGES <- TRUE

# Using a Boolean object
if (PACKAGES == TRUE) {
  install.packages(packages, dep = T)
}

# Which is the same as
if (PACKAGES) {
  install.packages(packages, dep = T)
}
```

Extra assignment

Create a function that

1. Takes as argument a vector of packages names
2. Loops through the packages listed in the input vector
3. Tests if a package is already installed
4. Only installs packages that are not yet installed
5. Loads the packages

- TIP: to test if a package is already installed, use the following code:

```
# Test if object x is contained in
# the vector of installed packages
x %in% installed.packages()
```

Using packages

Once a package is loaded, you can use its features and functions. Here's a list of some useful and cool packages:

- Rcmdr - Easy to use GUI
- swirl - An interactive learning environment for R and statistics.
- ggplot2 - beautiful and versatile graphics (the syntax is a pain, though)
- stargazer - awesome latex regression and summary statistics tables
- foreign - reads dtas and other formats from inferior statistical software
- zoo - time series and panel data manipulation useful functions
- data.table - some functions to deal with huge data sets
- sp and rgeos - spatial analysis
- multiwayvcov and sandwich - clustered and robust standard errors
- RODBC, RMySQL, RPostgresSQL, RSQLite - For relational databases and using SQL in R.

Resources

- A discussion of folder strucutre and data managament can be found here: https://dimewiki.worldbank.org/wiki/DataWork_Folder
- For a broader discussion of data management, go to https://dimewiki.worldbank.org/wiki/Data_Management

Git

Git is a version-control system for tracking changes in code and other text files. It is a great resource to include in your work flow.

We didn't cover it here because of time constraints, but below are some useful links, and DIME Analytics provides trainings on Git and GitHub, so keep an eye out for them.

- **DIME Analytics git page:**
  https://worldbank.github.io/dimeanalytics/git/
- **A Quick Introduction to Version Control with Git and GitHub:**
  https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004668

R projects

If you have used R before, you may have heard of RStudio Projects. It's RStudio suggested tool for workflow management. DIME Analytics has found that it is not the best fit for our needs, because

1. In DIME, we mainly use Stata, and we prefer to keep a similar structure in R (Stata 15 also has a projects feature, but it is not yet widely adopted)
2. We need to keep our code and data in separate folders, as we store code in GitHub and data in DropBox

However, if you want to learn more about it, we recommend starting here: https://r4ds.had.co.nz/workflow-projects.html